Process Models of Logic Programs: a Comparison

by
Annika Wærn

SICS R86007
Research Report
ISSN 0283-3638

Contents

1 Abstract

One way to deal with parallelism in logic programs is to define an execution model which is based upon communicating processes, instead of one sequential execution. Several such models have been given. This paper discusses what general properties such an execution model should fulfil, and gives a comparative study of some of the models proposed. Finally the approach of defining an operational model and distinguishing inherent parallelism in it is discussed briefly.

2 Introduction

2.1 Concurrency in Logic Programming

There are two main approaches on how to incorporate parallel execution schemes into hornclause program execution. The first approach is to concentrate on running hornclauses in parallel without adding any extra information. This approach gives rise to systems which need intelligent compilers and run-time systems to decide when parallelism can be used and when it should be avoided. Such systems have been described by /CoKi 81/ , /CoKi 84/ and /He 85/. The other approach is to leave some of these chores to be executed by the programmer, enriching the programming language with different non-logical control structure to control parallelism.

These approaches inherit their main execution order from Prolog - programs are executed top-down from one specific goal, for which a refutation is sought. It might be possible to define other execution schemes for programs, which are more useful in the parallel case than in sequential implementations. The scope of this paper is restricted to top-down execution, since the author is not aware of any alternative execution models of particular interest. If interesting alternatives are found, they could be subject to similar investigation.

2.2 And- and Or- Parallelism

The scope of this paper is to investigate process models for parallel execution of top-down resolution proofs on hornclauses. In this setting, one can distinguish two different states in which subproblems can be executed in parallel.

The first state is when there are several clauses which might be able to match a specific subgoal. All of these can be tried in parallel, but to give an answer, it suffices that one of them comes up with an answer. This is called or-parallelism. We will say that the clauses are tried 'in or-parallel'. The solutions given by the different clauses are independent, alternative solutions.
The second state is when there are several subgoals. These can be solved in parallel, but to give an answer, all of them must succeed and the answer substitutions must be compatible. Subgoals which share variables must therefore communicate their variable bindings to each other. This is called and-parallelism, and the term (used on subgoals) to be executed 'in and-parallel' will be used in the paper.

The use of parallelism on a true parallel computer ideally would have the effect of speeding up computations in direct proportion to the number of processors used, since operations that before were done sequentially would be executed simultaneously on different processors. But this result requires that all work done in the parallel setting really would be done in the sequential setting too. For logic programs, this is not nessecarily true. For example, if a subgoal fails, all other conjunctive subgoals need not to be tried. In the sequential setting, the subgoal most likely to fail is often executed first, reducing computation cost. In the parallel setting, all subgoals will start to execute immediately, giving rise to computations which were not executed at all in the sequential case. The same holds, and is perhaps a more serious problem, for or-parallel choices. Only one of the alternatives need to succeed, but all of them will be tried in parallel.

The notion of parallelism also has advantages which motivates pseudo-parallel implementations or use of coroutines. For example, the use of andparallelism can have the effect of reducing the number of alternatives that have to be sought, since the communication between subgoals allow more information to be present when the alternatives are explored. In combination with predicates giving sideeffects (such as read predicates), it is possible to write programs that only work when executed in or-parallel.

The ideal system would, of course, completely support both and- and or-parallelism. But unfortunately, such a system either results in great computational overhead or it gets very complicated. This problem is pictured over and over again in the process models described in this paper.

2.3 Why do We Want a Process Model ?

The definition of an appropriate theoretical process model is fundamental to the study of logic programs running in parallel systems:

- A process model is more or less naturally incrementable. A new system is obtained by adding a new process, and checking the resulting behavior of the system. Some process models lend themselves more easily to this than others, however.
- On a higher level, a process model could provide modularity, such that properties proven for a certain predicate, can be used when this predicate is combined with other predicates.
- A process model is a natural basis for discussing a true parallel implementation. Each process can be viewed as a basic unit, and the sceduling and routing of processes can be discussed on the basis on interprocess dependencies. Data internal to a process and data common to several processes can be distinguished and implemented in different ways.
- The notion of processes gives a very natural measurement of the amount of parallelism; namely the number of processes active in a certain situation. For theoretical reasons, a measurement of parallelism is interesting for two reasons; one is to discuss the amount of parallelism inherent in a certain program, another is the restrictions on the parallelism imposed by a control structure. For implementation issues, a good measurement of parallelism can make it possible to compare a computed theoretical optimum with runtime results.

One important use of a formal concurrency model is to serve as a mean to discuss the semantics of different parallel logic programming languages. Some of the control primitives introduced in such languages have the effect of obscuring the pure logic semantics of the language, in favor of a more procedural interpretation. A formal process model can serve as a mean to connect these two semantics more closely. The possibility to model different control structures of parallel logic programs is naturally crucial to this application.

In the following discussion of process models, we will concentrate on four properties:

- The ability to describe restriction operators of logic programming languages in the model
- The appropriateness of using the model for implementation discussions
- The appropriateness of the model as a measurement of parallelism
- The inherent incrementability and modularity of the model

3 Parallel logic programming languages

To discuss the appropriateness of process models for defining semantics of parallel logic programming languages, it is of interest how such languages generally are constructed. The opposite approach, building executional systems for nonrestricted hornclauses, is further discussed in the and-or process model part.

Parallel logic programming languages are primarily designed to make use of a very high level of and-parallelism. The or-parallelism is usually restricted or in some cases abandoned altogether. The main difference to the ancestor language Prolog is the introduction of new non-logical control structures, used mainly for controlling the parallelism. We can distinguish between three main approaches to do control parallelism, not nessecarily accomplished by different control structures.

- Sequencing subgoals or sets of subgoals. This can be accomplished by sequential and parallel and-operators, as in IC-prolog /ClCaGr 82/.
- Restricting or-parallelism, by switching from don't-know indeterminism to don't-care indeterminism in a certain situation and a certain subgoal, by *committing* to one of the possible clauses for a subgoal.
- Restricting the unifications allowed in a certain situation. This is usually done by annotating arguments to predicates as in Concurrent Prolog (where annotations can occur both in the head or the body of a clause), /Sh 83/, or using an input-output specification for each predicate, as is done in Parlog /ClGr 84/.

Examples:
The commit operator of Concurrent Prolog has three effects: Firstly it induces a sequence between two sets of subgoals, called the *guard* and the *body*, such that all subgoals of the guard must be executed before the commit operator is executed, which in its turn has to be executed before any one of the body predicates can be executed. Secondly the commit operator restricts or-parallelism by committing. Finally it takes care of the spreading of variable bindings from a local environment used during the or-parallel phase to the rest of the subgoals.

This spreading of variable bindings does not have any semantical effect in itself, but together with the read-only annotation of Concurrent Prolog, it works as a restriction on the possible unifications.

The trust operator used in Guarded Horn Clauses /Ue 85/ has also the effect of restricting the or-parallelism by committing. It can be executed whenever all guard predicates have succeeded. It does not induce any sequence over the guard and the body, however, and there exists no local environment during the or-parallel phase. Instead, unifications are restricted by classifying the variables of a clause in three groups, due to wether a variable has its first occurence in the head, the guard or the body. Unifications are then restricted differently depending on whether the trust operator has been executed or not. (This implies that the unification procedure can operate in two different modes depending on the current situation.)

The use of control primitives tend to obscure the pure logical semantics of hornclauses, in favor of a more procedural interpretation. This property of parallel logic programs can seem very unwished, but it has been argued that these control primitives serve a purpose in some important applications, like the implementation of operative systems. For high-level programming, a language more close to the logic semantics could be implemented on top of the parallel language, still keeping the benefits of using parallelism.

4 And- and Or- Processes

A process model frequently used is the and-or process model. It is closely related both the notion of and-or parallelism and to the and-or-trees, which are sometimes used for describing the semantics of a logic program at a semi-declarative level /Ko 79/. Parallel systems based on and-or-processes have been defined, see /CoKi 81/ and /CoKi 84/ for example. Such systems usually focus on running ordinary hornclauses (without extra control primitives) using or-parallelism and restricted and-parallelism.

4.1 Definition

In the and-or process model, we make use of two different types of processes, corresponding to the two modes of parallelism. The and-process requires that all of its subprocesses succeeds with consistent answer substitutions to succeed itself, while the or-process requires only one of its descendants to succeed. Modelling logic program execution this way, the initial goal corresponds to an and-process corresponding a conjunction of subgoals. For each subgoal, the and-process will invoke an or-process, corresponding to a disjunction of program clauses whose heads match the subgoal. These or-processes will, in turn, invoke one and-process for each such clause, corresponding to the body predicates of the clause. The procedure goes on recursively. The relation **invokes**(a,b) between processes forms the and-or-tree. The procedures are not dependent on any common data (except the program code), and all communication in between processes consists of messages, such as 'call', 'exit', 'redo' (if backtracking is used) and 'fail'. This property makes the semantics of the model very simple and has implementational advantages.

4.2 Incrementability

Since the invocation of processes is completely determined by the original goal, the model is not incrementable in the sense that a process can be added independently of other processes. Neither can modularity be achieved.

4.3 Defining Control Structure

The and-or process model is very well suited for modelling different *sequencing* schemes. In the parallel system suggested in /CoKi 84/, and-sequentiality is used as the regular mode, whereas and-parallellism can be specified in code, or deduced at runtime aided by input-output-specifications of predicates. Also, intelligent backtracking can be used on sequential parts in combination with and-parallelism.

Due to the structure of the and-or-trees, the effect of a *commitment operator* is easily defined. An operator of the cut-kind is executed in an and-process and kills all sibling and-processes, as well as any descendant or-processes that are invoked because of subgoals occuring to the left of the cut-operator. To perform correctly, the cut-operator should not be allowed to execute unless all such or-processes have succeeded at least once, giving consistent answer substitutions.

Also, the effect of *input-output relations* of the kind that requires ground instances of input arguments are easily described, and can be included in a execution system to allow the system to deduce the possible and-parallelism at runtime, as is done in /CoKi 84/. *Annotations* or other operators with similar effects are not so easily described using and-or-processes, however. This is due to the fact that these operators guide the concurrent processing and-parallel subgoals which share variables.

In the and-or-process model, the only check for shared variables is the consistency check, performed by the parent and-process for descendant or-processes. To model the effect of an annotation, an and-process needs to get information about current state of sibling and-processes. Such knowledge cannot be incorporated into the and-or process model, due to the fact that processes only communicate by messages and not by shared variables. The problem is pictured by the fact that implementations based on this process model normally only use restricted and-parallelism.

4.4 Measurement of Parallelism

In the and-or process model, the actual work performed by the system is not dependent on the number of processes, but on the number of messages transported in a certain situation. The number of processes active at one moment using and-or-processes can still be an accurate measurement on the amount of parallelism in a program, if an 'active process' is defined as a 'currently sending' process. This measurement is unfortunately difficult to establish. Approximational measurements can be found, defining 'active process' in a way which makes it easier to verify that a process is active.

4.5 Implementation

There are many possible way to define an executional system on and-or processes. The first choice is whether the generation of alternatives should be *eager* or *lazy*. If the eager mode is chosen, or-processes goes on immediately after reporting success, searching for alternative solutions. In the lazy mode, alternatives are only produced when required, that is, the system contains a backtracking scheme. The eager mode ia a simple implementation construction, but creates a great computational overhead. Another choice is to which amount and-parallelism should be used. Together with lazy generation of alternatives, complete and-parallelim becomes very complicated.

The fact that processes only use local variables and communicate by messages simplifies implementation, although the use of multiple environments creates overhead as compared to a sequential implementation.

4.6 Concluding Remarks on And-Or Processes

The and-or process model seems to be very well suited to discuss the effect of control structures regarding or-parallelism. Some of the control structures used in parallel logic programming languages to control and-parallelism are difficult to describe in this model though.

Finally, it can be noticed that the possible advantages of using a process model in theoretical discussions seem not to be present for the and-or process model. It is not incrementable or modular, it is difficult to define a proper measurement of parallelism in it, and it lacks information for being the basis for an implementation discussion.

5 Subgoal Processes

The notion of subgoal processes underlies the design of most parallel logic programming languages. The model can be viewed as the process model of and-parallelism in logic programming. It was introduced in /Ko 79/ as a strategy for execution of procedure calls.

5.1 Definition

In the subgoal process model, each subgoal is viewed as a process of its own. In a unification step, a process is reduced to a number of processes corresponding to the body of the clause. If the program clause is recursive, one of the recursive calls can be viewed as a continuation of the same process. Using this view, every predicate can be seen as a separate process, possibly invoking other processes.

This definition models the and-parallelity of programs, allowing subgoals to communicate using shared variables, a possibility which is not handled in the and-or process model. The model does not handle or-parallelism. In /Ko 79/ the control strategy is divided into two parts, the strategy for procedure call execution and the strategy for exploring alternatives. There is very little discussion of an execution model that allows the latter to be executed as a parallel search.

5.2 Incrementability

The model is incrementable since adding a new process simply corresponds to adding a new subgoal. Modularity is more difficult to establish however, since the model does not provide any information about a program part without a corresponding goal.

5.3 Defining Control Structure

Most of the control structures used in parallel logic programming languages are very easily described using the subgoal process model. This comes naturally since this process model underlies the construction of these languages. The model is not sufficient to discuss the control structures handling or-parallelism, since it does not handle or-parallel choices. Some control structure behaviors that can be described cannot be justified in the model, due to the same cause. Some examples:

- Annotations have the effect of limiting the number of unifications allowed. In the subgoal process model, this corresponds to limiting the number of process reductions allowed from a certain process. The same holds for input-output relations.
- The unification restrictions used in Guarded Horn Clauses have the same effects as annotations. Restrictions change when the trust operator is executed however, and this change cannot be decribed nor justified using the subgoal process model.
- The effect of commitment operators cannot be described using subgoal processes.

5.4 Measurement of Parallelism

The number of active processes is in the subgoal process model provides an easily established measurement of parallelism. To be accurate, or-parallel alternatives have to be handled as well. It could be discussed whether or-parallel alternatives for a goal should be counted as one process, since only one of them will contribute to the solution. But since they correspond to parallel search of several alternatives, they increase efficiency as compared to sequential search, as long as the alternatives would be sought in a sequential execution system as well.

5.5 Implementation

The subgoal process model lends itself quite well for implementational discussions. It does not suffice for describing the complete behavior of an executional system, since information about how to handle the or-parallellism is needed.

5.6 Concluding Remarks on Subgoal Processes

The subgoal process model is useful for discussing some, if not all, implementation issues, but the notion seem to weak to discuss theoretical problems. A useful model must handle both and-parallelism and or-parallelism with the same strength.

6 Program Clause Processes

In the program clause model, each clause is seen as a procedure, which when invoked gives rise to a process. The approach has been quite thoroughly investigated by R. Gustavsson, L. Beckman and the author (main reference /BeGuWae 86/) using a formalism based on the CCS calculus for communicating processes /Mi 80/, /Mi 82/. This chapter of the paper will refer to this model.

6.1 Definition

Each clause is translated to a certain kind of CCS-expression, which allows all of the body atoms to be accessed in parallel with the reusage of the clause, but only after the head atom has been accessed. (This definition restricts the model to top-down execution.) The representation of a hornclause program is a CCS-expression which combines the representation of the clauses such that all clauses can communicate with each other. Parallelism is modelled by interleaving, that is, when two processes act concurrently, it is interpreted as if either one of them executes first. Every communication can, using this particular restricted use of CCS, be proven to correspond to a resolution step. The resulting system can, given an appropriate interpretation, be proven to be a sound and complete inference system on horn clauses /BeGuWae 86/, /Wae 86/.

An interesting fact about this process model is that it can give a semantic interpretation both for a program on its own, and for a program together with a goal. From an executional point of view, only the latter is interesting, but the earlier forms a nice connection to the declarative semantics of a hornclause program.

The interpretation of the model is based on the set of possible communications in each situation. This relation between situations can be pictured as a dependency graph, using the relation (sit1 **transforms to** sit2 **in one step**). The graph might not seem to be a proper tree, since a situation identical to an earlier one can occur after a series of transformations. However, it is not nessecary to view it as the *same* situation, and the graph therefore can be viewed as forming a rooted tree, where the initial situation is the root. From each possible situation the current goal can be extracted. In the model described by the references, the goal represented is indeed the only thing that distinguish situations from each other. (It is possible to formulate models where more information is kept.) If the nodes of the tree is set to represent the current goal instead of the complete situation, the relation tree turns into a complete top-down resolution tree.

6.2 Incrementability

The model is incrementable in the sense that a new procedure corresponding to a clause always can be added, and the effects in forms of new possible transformations can be studied. (The extension is monotone as long as the system is restricted to proper horn clauses.)

On the predicate level the system is modular in the same way as the ordinary declarative semantics of horn clauses is modular. Properties for a single predicate hold when the predicate is combined with other predicates only when the predicate does not use the other predicates as subroutines. To get modularity at the predicate level, programs have to be verified bottom-up.

On a higher level, where groups of predicates are combined into modules, the CCS formalism allows true modularity. The formalism can be used to restrict which predicates are allowed to be exported from a module. The transformation graphs can also be used to determine exactly what predicates need to be imported to a module. Thus, the interaction between modules can be restricted and verified to work correctly. Another property of the formalism is that the representation can be changed to allow bottom-up resolution instead of top-down resolution. It is thus possible to combine modules which use top-down resolution with modules that use bottom-up resolution and verify that they execute correctly together /Gu 86/.

6.3 Defining Control Structure

In the model here described, there are three different ways to describe control structure. The first is to use the original CCS operators to describe the behavior of a certain control structure. Using the CCS representation chosen in the references, this approach is not possible. Another approach is to redefine the axiom of the transition relation to take these restrictions in account, and then enrich the language of the agents to contain the control structure. This approach is used in /Be 86/ and seems to work out well for several control structures affecting and-parallelism. The last approach is to describe the effects on the execution trees, by pruning or adding branches. This approach is neccesary for describing commitment operators.

In /Be 86/ it is showed how *annotations* and *input-output relations* can be described as restrictions on the axioms of the transition relation of CCS. The effect of a sequence operator is unfortunately not so easy described, due to that subgoals appearing as the result of a resolution step is not syntactically placed in the place of the subgoal. In the model it is possible to define that the refutation of one subgoal should start before another subgoal, but the requirement that it should be completely finished is difficult to picture. This fact does also effect the possibility of representing commitment operators, since these only should be allowed to execute when all guard predicates have succeeded.

It has already been mentioned, that to decribe the effect of *committing* to an alternative it is nessecary to look at the execution trees. Unfortunately, the full top-down resolution trees contain too much information. From each node of the tree, branches corresponding to different ways to resolve the same subgoal against the program exists, as well as branches corresponding to starting with different subgoals. The first kind of branches are neccesary, that is, to get a complete execution system they all have to be tried. The second kind of branches are not nessecary and will only produce additional copies of the same solutions. In a normal execution system, only the first kind of branches are represented, the other kind of branches are implicitly chosen by the order in which and-processes are executed. Since both kind of branches are kept in the complete top-down resolution tree and there is no way to distinguish between them, it is not possible to determinate which branches a commitment really prunes.

6.4 Measurement of Parallelism

When the idea of programs as CCS processes originally was formed, the thought was that the number of processes should be constant and determined at compile time. Each process contains then one program clause, as well as all presently executing invocations of that clause. This approach gives no measurement of the amount of parallelism in a program. Another approach is to view every invocation of a clause as a process. This idea gives a measurement of parallelism that grows slower than the number of subgoal processes, but is otherwise very closely related to that measurement. It does not measure the amount of or-parallelism.

Another approach is to view every node in the transformation tree as an independent process. Using this view, a process terminates after one step, giving rise to several new processes, one for each of its immediate descendant nodes. In a sense, the number of such processes measure both the amount of and-parallelism and the amount of or-parallelism, but is not possible to tell how much of which sort. The number of nodes in the tree also grows very rapidly.

6.5 Implementation

The process model here described cannot directly be the basis for an implementation, but the transformation trees can, using the view of nodes as processes. Some suggestions concerning such an implementation are mentioned in /Wae 86/. There are two important drawbacks of the approach though. The first is due to the fact that the program clause process model simulates full and- and or-parallelism. This makes it neccesary to represent each state alternative together with a binding environment of its own. The overhead gets even worse from the fact that and-parallelism is modelled by interleaving, and the model thus gives rise to an enormous amount of unneccesary alternatives. The execution system itself can be kept very simple though, which makes it possible to concentrate on different schemes for reducing the overhead.

7 Execution Trees: an Operational Model

As mentioned earlier, it is nessecary to discuss execution trees when modelling the behavior of operators restriction the or-parallelism. And-or-trees do not picture the true behavior of a normal execution system, since they do not contain any information about in which order the branches are sought. This part introduces therefore a kind of trees, in which the execution order of subgoals is pictured, however not the search order over resolution alternatives for a subgoal. They can be viewed as an operational model rather than a process model. To motivate parallel implementations from this kind of model, it is nessecary to distinguish inherent parallelity in it. The definition of the trees is a generalization of SLD-trees, as described in /Llo 84/. The model presupposes immediate propagation of values, which is not necessarily true in a distributed implementation.

7.1 Definition

In SLD-resolution, only a restricted set of resolution steps are allowed to be used. The following restrictions are used:
- Resolution proofs are executed top-down.
- Only one subgoal can be resolved at a time. The subgoal used is determined by a selector function, which from a subgoal set picks one subgoal.

For each possible goal and each selector function one SLD-resolution tree is formed.

To model parallel behavior, we need to losen up the restrictions imposed in SLD-resolution. Firstly, the selector function need not be a function at all. In identical situations, the same subgoal need not to be chosen. Secondly, several subgoals can resolve in the same moment. The second is not nessecary to model, since parallelism can be modelled by interleaving, as is done in the CCS model described earlier. It is incorporated here to get a picture of true parallel behavior. The following restrictions hold:

- Resolution proofs are executed top-down.
- The subgoals chosen to resolve one step in parallel are related by the subset relation on the set of subgoals.

Parallel resolution steps need to fulfil a new restriction to succeed. All resolutions made in the same resolution step need to give compatible substitutions, that is, they should not give rise to circular structures or instantiate a variable to different values.

Each choice of subset in a node corresponds to a new tree. The SLD-trees are a proper subset of these trees. The number of possible trees gets very high, in some cases uncountable.

7.2 Incrementability

An execution tree can be viewed as a process model, in the same way as the complete top-down trees could for the CCS model. But then the set of processes is completely determined by the original goal and the goal subsets chosen. The model is not incrementable or modular.

7.3 Defining Control Structure

The effect of control structure can in this model be described using two different approaches. The first approach, which applies to restricting and-parallelism, is to restrict what processes can be chosen to participate in a parallel resolution step. The subset chosen should then be related to the set of current subgoals by a relation which is more restrictive than the subset relation. The second approach, used for restricting or-parallelism, is to directly describe what effects a restriction operator would have on the search tree, in the sense of cutting away already existing branches.

Annotations on variables would be modelled by a generally described restriction of the subset relation, describing when an annotated variable can be allowed to unify, mirroring the restrictions in the unification algorithm.

Input-output relations would be modelled by restrictions in the subset relation, which would be specific to each predicate.

For *sequencing*, the restrictions in the subset relation would be that any subgoal to the left of a sequence operator has to be executed before the sequence operator, and every subgoal to the right of the sequence operator has to wait until the sequence operator has been executed. To work properly, every resolution step has to construct a resolvent in which the new subgoals are placed in the place of the subgoal resolved.

A *commitment operator* cuts away branches in the search tree when executed. It is not trivial to say which branches are cut away, but it is possible since it is completely known to the tree which subgoal that has been resolved upon in every step. The chore is complicated in the case when several subgoals have executed in the same step. If one of them commits, only the branches corresponding to its alternatives should be cut away. The branches emerging from such a node correspond to the cartesian product of solutions for the subgoals. The ones corresponding to alternatives in the restricted one are distinguishable, but this is a complicated operation.

7.4 Measurement of Parallelism and Implementation

There is no immediate measurement of parallelism in this model, since it is not based on processes. To define any parallel implementation based on this kind of model, processes have to be extracted from it by recognizing inherent parallelism in it. One such source of parallelism is, as mentioned earlier, to view every alternative as a process. It is more difficult though to recognize sources of and-parallelism.

8 Conclusions and Future Work

An efficient process model could provide a good theoretical framework for discussing properties of logic programming in a parallel environment. It seems though as if the currently suggested process models have been defined from recognizing sources of parallelism and describing them in a process model. This makes them unsuitable to describe sources of parallelism that were not originally discovered. A more fruitful approach for achieving a general theoretical model seems to be to define a general operational model and using a general definition of parallelism to distinguish sources of parallelism in it. The aim of the auhor is to make some work in this direction.

9 References

/BeGuWae 86/ **Beckman L. Gustavsson R. & Waern A .**
'An Algebraic Model of Parallel Execution of Logic Programs'
In *Proc. for Symposium on Logic in computer science,*
IEEE Computer society press, ISBN 0-8186-0720-3 (1986)
p. 50-57.

/Be 86/ **Beckman L.**
'Towards a Formal Semantics for Concurrent Logic Programming Languages'
Proc. third international Conf. on Logic Programming, LNCS
Vol 225 Springer Verlag, ISBN 3-540-16492-8 (1986)
p. 335 - 349.

/ClCaGr 82/ **Clark K. L., McCabe F.G & Gregory S.**
'IC-Prolog Language Features'
in *Logic Programming* Academic Press, ISBN 0-12-175520-7
(1982) p. 253 - 266.

/ClGr 84/ **Clark K. & Gregory S.**
'PARLOG: Parallel Programming in Logic'
IC Research report DOC 84/4 (1984).

/CoKi 81/ **Conery J. S. & Kibler D. F.**
'Parallel Interpretation of Logic Programs'
ACM (1981).

/CoKi 84/ **Conery J. S. & Kibler D. F.**
'AND Parallelism and Nondeterminism in Logic Programs'
New Generation Computing.3 (1985).

/Gu 87/ **Gustavsson R.**
'Parallelism and Logic Programming'
Forthcoming as SICS report (1987).

/He 85/ **Hermenegildo M.V.**
'A Restricted AND-Parallel Execution Model and Abstract Machine for Prolog Programs'
MCCS Tech. rep. PP-104-85 (1985).

/Ko 74/ **Kowalski R.**
'Predicate Logic as a Programming Language'
Proc. IFIPS (1974).

/Ko 79/ **Kowalski R.**
'Algorithm = Logic + Control'
CACM Vol 22 , No 7 (1979).

/Ko 79/ **Kowalski R.**
Logic for problem solving
Artificial Intelligence Series, North-holland ISBN 0-444-00365-7 (1979).

/Llo 84/ **Lloyd, J.W.**
Foundations of Logic Programming
Springer Verlag ISBN 3-540-13299-6 (1984).

/Mi 80/ **Milner R.**
A Calculus of Communicating Systems
LNCS Vol 92, Springer Verlag (1980).

/Mi 82/ **Milner R.**
'Calculi for Synchrony and Asynchrony'
University of Edinburgh, Dept. of Computer Science, internal report CSR-104-82 (1982).

/Sh 83/ **Shapiro E.Y.,**
'A Subset of Concurrent Prolog and Its Interpeter'
Weizmann Institute of Science TR-003 (1983).

/Ue 85/ **Ueda K.,**
'Guarded Horn Clauses'
ICOT Technical Report TR-103 (1985).

/Wæ 86/ **Wærn A.,**
'A Computational Model for parallel execution of logic programs'
Internal UPMAIL document (1986).